2008

Denotational style correctness of a CPS-Transform based compiler

Arthur Nunes-Harwitt

Denotational Style Correctness of a CPS-Transform Based Compiler

Arthur Nunes-Harwitt
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, New York 14623
anh@cs.rit.edu

ABSTRACT

Correctness is a crucial property for compilers; programmers rely on it when writing code. Ideally, correctness should be proved. Work on compiler correctness has focused on direct translation strategies. However, in practice, the continuation passing style (CPS) transform (or a variant) is often used in the translation process. Here a simple source language and its CPS-transform based compiler are introduced. A tractable proof for this compiler is presented, including a denotational proof of the correctness of a CPS-transform. The benefits of the proof are discussed.

1. INTRODUCTION

Compilers translate source code to target code. Today, most programs are written using a high-level language that is compiled. It is widely agreed[1] that compilers must be correct, since otherwise the generated code is not reliable. A formal correctness proof bolsters one's confidence that a compiler generates target code that does what is specified by the source code.

Typically, unless explicitly stated otherwise, "compiler correctness" focuses on the middle-end of the compiler: the part that accepts an abstract syntax tree and generates a linear (or almost linear) intermediate-language consisting of relatively low-level instructions which can be easily translated into the machine language of choice. There are two approaches when writing a middle-end: direct and staged. The *direct* approach translates the source to the target directly without any intermediate transformations, whereas a *staged* approach breaks the translation into n stages (where $n > 1$). Each stage transforms its input into output suitable for the next stage's input; the output language may be different from the input language or the same.

Compiler correctness has been studied extensively using both operational[11, 19] and denotational techniques[23, 4, 16, 17]. And while separate back-end stages have been studied[16, 17], the middle-end is invariably studied with a direct approach. In practice, however, the middle-end is often staged. In particular, many middle-ends make use of the CPS-transform[18, 10], or a variation such as the A-normal form (ANF) transform[9], as an initial stage when generating output (e.g. Scheme[12, 21] implementations such as Rabbit[22], Orbit[13], Twobit/Larceny[5], DrScheme[8], Chicken[3]).

This paper defines a three-staged compiler (middle-end) that makes use of the CPS-transform; it models the staged middle-ends used in practice. Compiler correctness is proved using denotational techniques. Section 2 defines the source language. Section 3 defines the target language. Section 4 defines the intermediate CPS language. Section 5 discusses linearization, the CPS-transform, a normalization stage, and the stage that translates from the intermediate language to the target code. Section 6 discusses the correctness theorems; details of the proofs are in the appendix. Section 7 concludes and mentions possible future work.

2. SOURCE LANGUAGE

The source language is the call-by-value λ-calculus with constants. These constants include numbers as well as operators that act on the numbers, such as addition. Abstractions model user-defined procedures, and applications model procedure calls. Note that there are a number of layers of syntax defined. Values must be defined as a distinct subclass of terms. Another useful subclass is $\mathcal{W}$, which is the set of atomic values.

Syntax

$$
\begin{array}{c}
c_{op} \in \text{Operators} \\
c \in \text{Constants} \supset \text{Operators} \\
x \in \text{Variables} \\
R \in \text{Result} ::= c \mid (\lambda x.M) \\
W \in \mathcal{W} ::= c \mid x \\
V \in \text{Values} ::= W \mid (\lambda x.M) \\
M, N \in \Lambda ::= V \mid (M\ N)
\end{array}
$$

Evaluation contexts are defined below. A context is an expressions with a hole in it, where the hole is written []. As the name suggests, the term that is to be evaluated next fills the hole of an evaluation context. Evaluation contexts are needed in section 5.

$$\mathcal{E} ::= [] \mid \mathcal{E}[(V\ [])] \mid \mathcal{E}[([]\ M)]$$
or $\mathcal{E} ::= [] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M)$

What's missing?

- Procedures with multiple parameters
  These can be simulated using Currying.
- A conditional expression such as 'if'
  This expression can be simulated using constant operators and thunks. In fact, if one understands the Clinger-Hansen[6] dictum "lambda is label" to be bi-directional, and since 'if' is implemented using labels, then perhaps one ought to understand 'if' this way.
- Recursion
  Recursion can be implemented using the Y-operator which can be represented in this language.
- Looping constructs
  Looping can be implemented in terms of recursion.
- Assignment, exceptions, etc.
  This paper will not deal with such features.

There are several ways of defining the semantics of the language above. In particular, a denotational semantics gives a meaning to each kind of term based on the meaning of each of its sub-terms. While this method may seem indirect, since the meaning is expressed in terms of a meta-language, the advantage is that the meta-language is designed for proving equations.

Denotational Semantics

$$\begin{array}{lcl}
\mathcal{D}[\![c]\!] & = & \lambda\rho.\lambda\kappa.(\kappa\ \mathcal{K}(c)) \\
\mathcal{D}[\![x]\!] & = & \lambda\rho.\lambda\kappa.(\kappa\ \rho(x)) \\
\mathcal{D}[\![(\lambda x.M)]\!] & = & \lambda\rho.\lambda\kappa.(\kappa\ (\lambda v.\mathcal{D}[\![M]\!]\rho[x \mapsto v])) \\
\mathcal{D}[\![(M\ N)]\!] & = & \lambda\rho.\lambda\kappa.\mathcal{D}[\![M]\!]\rho(\lambda m.\mathcal{D}[\![N]\!]\rho(\lambda n.((m\ n)\ \kappa)))
\end{array}$$

$$\begin{array}{lcl}
\rho & \in & \text{Variables} \to E \\
\kappa & \in & E \to A \\
\mathcal{K} & \in & \text{Values} \to E
\end{array}$$

E is a semantic value
A is an answer

The function ρ serves as the environment mapping variables to semantic values, κ serves as the continuation function mapping semantic values to answers, where an answer might be a semantic value or a coarser representation of one, and the function $\mathcal{K}$ maps constants to semantic values. These definitions are given for the intuition they provide. Neither the domains E and A nor the details of the functions ρ and κ are crucial for the proofs, so these details will be ignored.

3. TARGET LANGUAGE

The target language is meant to resemble the instruction set of a load-store machine architecture. For example, both `load` c, x and `move` x, z instructions are very similar to what one finds on a load-store machine[1]. The instruction `instr:`$c\ x, z$ may look unfamiliar, but that is only because it represents a family of instructions. For example, `add1` x, z (which adds one to the contents of x and puts the result in z) would be expressed by `instr:succ` x, z, and `sub1` x, z (which subtracts one from the contents of x and puts the result in z) would be expressed by `instr:pred` x, z. Other instructions are more complex; for example, a single instruction `call` y, x, z creates an activation record, and then passes the contents of x to the procedure y refers to. The instruction `makeClosure` $x, \{S\}, y$ is also complex; given the formal parameter x and the code sequence S, it creates a closure and puts it in y. Further `makeClosure` $x, \{S\}, y$ has internal structure: the instruction is part of a code sequence but also contains S, which is a code sequence. I have in mind that the internal structure represents a pointer; to emphasize this I introduce additional notation and wrap the internal sequence in braces. Furthermore, although this sort of internal structure is easy to eliminate, I leave it in because it makes reasoning about the language simpler.

Syntax

$c \in$ Constants
$x, y, z \in$ Variables

$$\begin{array}{l}
I \in \text{Instructions} ::= \\
\quad \texttt{load}\ c, x \quad \mid \quad \texttt{move}\ x, z \\
\quad \mid \texttt{makeClosure}\ x, \{S\}, y \quad \mid \quad \texttt{instr:}c\ x, z \\
\quad \mid \texttt{call}\ y, x, z
\end{array}$$

$$S \in \text{Sequence} ::= \texttt{return}\ x \mid \texttt{tailCall}\ y, x \mid I; S$$

Denotational Semantics

$$\begin{array}{ll}
\mathcal{D}_s[\![\texttt{return}\ x]\!] & = \lambda\rho.\lambda\kappa.(\kappa\ \rho(x)) \\
\mathcal{D}_s[\![\texttt{tailCall}\ y, x]\!] & = \lambda\rho.\lambda\kappa.((\rho(y)\ \rho(x))\ \kappa) \\
\mathcal{D}_s[\![\texttt{load}\ c, x; S]\!] & = \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![S]\!]\rho[x \mapsto \mathcal{K}(c)]\kappa \\
\mathcal{D}_s[\![\texttt{move}\ x, z; S]\!] & = \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![S]\!]\rho[z \mapsto \rho(x)]\kappa \\
\mathcal{D}_s[\![\texttt{makeClosure}\ x, \{S'\}, y; S]\!] & = \\
\quad \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![S]\!]\rho[y \mapsto (\lambda v.\mathcal{D}_s[\![S']\!]\rho[x \mapsto v])]\kappa & \\
\mathcal{D}_s[\![\texttt{instr:}c\ x, z; S]\!] & = \\
\quad \lambda\rho.\lambda\kappa.((\mathcal{K}(c)\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![S]\!]\rho[z \mapsto v]\kappa)) & \\
\mathcal{D}_s[\![\texttt{call}\ y, x, z; S]\!] & = \\
\quad \lambda\rho.\lambda\kappa.((\rho(y)\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![S]\!]\rho[z \mapsto v]\kappa)) &
\end{array}$$

4. INTERMEDIATE LANGUAGE

Many compilers work by first translating the source code into an intermediate language, which is often the CPS-language[2].

[1]The key difference being that the semantics below makes x and z environment variables rather than registers. Nevertheless, I am imagining that registers would be used to implement the environment.

Early approaches to using the CPS-language as an intermediate language suffered from two problems: (1) it wasn't clear how to distinguish between abstractions representing user defined procedures and abstractions representing continuations, and (2) CPS-transforms introduced many administrative reductions. I make use of the Sabry-Wadler CPS-language[20], which solves the first problem by clearly defining a CPS-language in which abstractions and continuations are syntactically distinct. Their CPS-transform (and variations of it) also introduce no administrative reductions, thus solving the second problem. I will say more about the CPS-transform in the next section. There are three syntactic categories in the CPS-language below: Values, Continuations, and terms. Values are similar to those in the source language; however, here abstractions take an explicit continuation parameter. Continuations can be either a continuation variable, or a continuation procedure. Note that k does *not* refer to a class of continuation variables, but rather to a single continuation variable. Terms are either continuations applied to values, or an application of values applied to a continuation; in contrast to the source language, a term cannot be a value.

Syntax

$$V_c \in \text{Values}_c ::= W \mid (\lambda x.\lambda k.M_c)$$
$$Q_c \in \text{Continuations} ::= k \mid (\lambda x.M_c)$$
$$M_c, N_c \in \Lambda_c ::= (Q_c\ V_c) \mid ((V_c\ V_c')\ Q_c)$$
$$R_c \in \text{Result}_c ::= c \mid (\lambda x.\lambda k.M_c)$$

Denotational Semantics

$$\begin{array}{ll}
\mathcal{D}_c[\![(Q_c\ V_c)]\!] & = \lambda\rho.\lambda\kappa.((\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)\ (\mathcal{D}_V[\![V_c]\!]\rho)) \\
\mathcal{D}_c[\![((V_c\ V_c')\ Q_c)]\!] = & \\
\quad \lambda\rho.\lambda\kappa.(((\mathcal{D}_V[\![V_c]\!]\rho)\ (\mathcal{D}_V[\![V_c']\!]\rho))\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)) & \\
\mathcal{D}_V[\![c]\!] & = \lambda\rho.\mathcal{K}(c) \\
\mathcal{D}_V[\![x]\!] & = \lambda\rho.\rho(x) \\
\mathcal{D}_V[\![(\lambda x.\lambda k.M_c)]\!] & = \lambda\rho.(\lambda v.\mathcal{D}_c[\![M_c]\!]\rho[x \mapsto v]) \\
\mathcal{D}_Q[\![k]\!] & = \lambda\rho.\lambda\kappa.(\lambda v.(\kappa\ v)) \\
\mathcal{D}_Q[\![(\lambda x.M_c)]\!] & = \lambda\rho.\lambda\kappa.(\lambda v.\mathcal{D}_c[\![M_c]\!]\rho[x \mapsto v]\kappa)
\end{array}$$

5. COMPILATION

Winston and Horn in the third edition of their book *LISP*[24] write that "Compilers are usually major undertakings." In fact, compiling to Scheme is no more difficult than writing an interpreter in Scheme, provided the interpreter has been written in a compositional style. Nevertheless, I agree that writing even a naive compiler can be difficult. Why? Compiling typically involves translating from a nested language, such as the source language above, to a non-nested linear language; it is the process of transforming a nested language into a linear one that is tricky.

Nesting in the source language is caused by two constructs: abstractions and applications. Abstractions are not hard to linearize. It's the applications that make for the subtlety. There are two approaches to linearizing applications: direct and staged.

5.1 Direct Linearization

The direct approach translates the source to a linear form without any intermediate transformations. A local translation is frequently used for direct linearization[4, 16, 17, 14]; however, a CPS based translation is also possible[23].

5.1.1 Local

Given an abstract syntax tree, local linearization implicitly involves the following four ideas.

1. Placing an explicit call operator at the end of an application and moving constant operators to the end of an application do not affect the semantics.
2. If we add a marker to do a debug-step operation[15], we see that there are only results to the left of the marker. Further, taking a step either involves moving into an application where the marker ultimately moves over a value, which looks like a push, or reducing an application where some of the values to the left of the marker are eliminated, which looks like a pop.
3. Because the arity of constant operators in known, and the arity of others can be recorded with the call operator, the form from (1) can be written with little or no nesting (in a linear form).
4. Because of (2), values can be translated into push instructions for a stack-machine.

Consider the following example with the abstract-syntax represented as an s-expression: `(f 2 3 (+ 4 5))`.
`(f 2 3 (4 5 +) call`^{4}`)` is an equivalent representation.
`f 2 3 4 5 + call`^{4} is also equivalent.
`push f; pushi 2; pushi 3; pushi 4; pushi 5; add; call 4` is simply another variation.

These ideas are easily combined into a local linearization algorithm. An example local linearization is given below for the source language from section 2.

$$\begin{array}{lcl}
\mathcal{L}[\![c]\!] & = & \texttt{pushi}\ c \\
\mathcal{L}[\![x]\!] & = & \texttt{push}\ x \\
\mathcal{L}[\![(\lambda x.M)]\!] & = & \texttt{pushClosure}\ x, \{\mathcal{L}[\![M]\!]\} \\
\mathcal{L}[\![(c\ M)]\!] & = & \mathcal{L}[\![M]\!]; L_{const}[\![c]\!] \\
\mathcal{L}[\![(M\ N)]\!] & = & \mathcal{L}[\![M]\!]; \mathcal{L}[\![N]\!]; \texttt{call}^2 \text{ if } M \neq c
\end{array}$$

5.1.2 CPS

It is possible to use continuation passing style and higher-order assembly language (HOAL)[23] to develop a non-local linearization algorithm. K is the syntactic continuation called the *sequel*; the initial sequel is $(\lambda v.\texttt{halt}\ v)$. An example direct CPS linearization is given below for the source language from section 2; the target is the HOAL variation of the target language presented in section 3.

$$\begin{array}{lcl}
\mathcal{C}[\![c]\!]K & = & \texttt{load}\ c\ K \\
\mathcal{C}[\![x]\!]K & = & \texttt{move}\ x\ K \\
\mathcal{C}[\![(\lambda x.M)]\!]K & = & \texttt{makeClosure}(\lambda x\kappa.\mathcal{C}[\![M]\!](\lambda v.\texttt{return}\ \kappa\ v))\ K \\
\mathcal{C}[\![(M\ N)]\!]K & = & \mathcal{C}[\![M]\!](\lambda y.\mathcal{C}[\![N]\!](\lambda x.\texttt{call}\ y\ x\ K))
\end{array}$$

5.2 Staged Linearization

The staged approach consists of stages such that each stage translates an input form to an output form. For the initial stage, the input form is the source language; for the final stage the output is the target language. Frequently, the CPS-transform (or a variant such as the ANF-transform) is used as the initial stage.

I'll characterize CPS-like transforms by giving an intuitive description of the ANF-transform; however, the compiler given below will make use of the CPS version.

ANF linearization involves two key ideas: (1) There is exactly one application that will be evaluated first, and (2) this expression can be pulled out, and its value named without changing the meaning of the expression.

Consider the following example with the abstract-syntax represented as an s-expression: `(f 2 3 (+ 4 5))`. The application that will be evaluated first is `(+ 4 5)`.
Thus the code is rewritten as `(let ((x (+ 4 5))) (f 2 3 x))`. One can imagine this form being translated into the the following: `add 4, 5, x; tailCall f, 2, 3, x`.

An example ANF-transform is given below for the source language from section 2, where $\mathcal{E}$ is the context in which the application to be evaluated first is found[9].

$$\begin{array}{lcl}
\mathcal{A}[\![V]\!] & = & V^{\#} \\
\mathcal{A}[\![(V\ V')]\!] & = & (V^{\#}\ V'^{\#}) \\
\mathcal{A}[\![\mathcal{E}[(V\ V')]]\!] & = & \mathsf{let}\ x = (V^{\#}\ V'^{\#})\ \mathsf{in}\ \mathcal{A}[\![\mathcal{E}[x]]\!] \\
 & & \text{if } \mathcal{E} \neq [] \text{ and } x \text{ is fresh}
\end{array}$$

$$\begin{array}{lcl}
W^{\#} & = & W \\
(\lambda x.M)^{\#} & = & (\lambda x.\mathcal{A}[\![M]\!])
\end{array}$$

The output form of $\mathcal{A}$ is discussed by Sabry and Wadler[20]. It turns out, that it is isomorphic to the CPS-language. Thus, the transform above is easily modified so as to generate the intermediate language in section 4.

$$\begin{array}{lcl}
\mathcal{A}'[\![V]\!] & = & (k\ V^{*}) \\
\mathcal{A}'[\![(V\ V')]\!] & = & ((V^{*}\ V'^{*})\ k) \\
\mathcal{A}'[\![\mathcal{E}[(V\ V')]]\!] & = & ((V^{*}\ V'^{*})\ (\lambda x.\mathcal{A}'[\![\mathcal{E}[x]]\!])) \\
 & & \text{if } \mathcal{E} \neq [] \text{ and } x \text{ is fresh}
\end{array}$$

$$\begin{array}{lcl}
W^{*} & = & W \\
(\lambda x.M)^{*} & = & (\lambda x.\lambda k.\mathcal{A}'[\![M]\!])
\end{array}$$

$\mathcal{A}'$ is in fact the CPS-transform. In practice, this algorithm is inefficient; an efficient first-order one-pass CPS-transform by Danvy and Nielsen[7] is given below.

$$P \in \Lambda \setminus \text{Values}$$

$$\begin{array}{lcll}
\mathcal{F}[\![V]\!]Q_c & = & (Q_c\ V^{\star}) & \\
\mathcal{F}[\![(V\ V')]\!]Q_c & = & ((V^{\star}\ V'^{\star})\ Q_c) & \\
\mathcal{F}[\![(V\ P)]\!]Q_c & = & \mathcal{F}[\![P]\!](\lambda x.((V^{\star}\ x)\ Q_c)) & \\
 & & x \text{ is fresh} & \\
\mathcal{F}[\![(P\ V)]\!]Q_c & = & \mathcal{F}[\![P]\!](\lambda y.((y\ V^{\star})\ Q_c)) & \\
 & & y \text{ is fresh} & \\
\mathcal{F}[\![(P\ P')]\!]Q_c & = & \mathcal{F}[\![P]\!](\lambda y.\mathcal{F}[\![P']\!](\lambda x.((y\ x)\ Q_c))) & \\
 & & x \text{ and } y \text{ are fresh} &
\end{array}$$

$$\begin{array}{lcl}
W^{\star} & = & W \\
(\lambda x.M)^{\star} & = & (\lambda x.\lambda k.\mathcal{F}[\![M]\!]k)
\end{array}$$

5.3 The Staged Compiler

The first stage of the compiler is $\mathcal{F}$ which does a CPS-transform of the source language; its output is the mostly linear CPS-language. Note that since $\mathcal{F}$ requires a continuation parameter to compute its result, it is supplied an initial continuation parameter: the fixed continuation variable k.

The second stage of the compiler is $\mathcal{N}$ (defined below) which normalizes the CPS-language code by naming constants and abstractions; the output is a subset of the CPS-language. The next stage would require a longer case analysis for its definition without this normalization stage.

$$\begin{array}{lcll}
\mathcal{N}[\![(k\ x)]\!] & = & (k\ x) & \\
\mathcal{N}[\![(k\ R_c)]\!] & = & ((\lambda x.(k\ x))\ R_c^{\dagger}) & x \text{ is fresh} \\
\mathcal{N}[\![((\lambda x.M_c)\ V_c)]\!] & = & ((\lambda x.\mathcal{N}[\![M_c]\!])\ V_c^{\dagger}) & x \text{ is fresh} \\
\mathcal{N}[\![((V\ x)\ Q_c)]\!] & = & ((V^{\dagger}\ x)\ Q_c^{\ddagger}) & \\
\mathcal{N}[\![((V\ R_c)\ Q_c)]\!] & = & ((\lambda x.((V^{\dagger}\ x)\ Q_c^{\ddagger}))\ R_c^{\dagger}) & x \text{ is fresh}
\end{array}$$

$$\begin{array}{lcl}
W^{\dagger} & = & W \\
(\lambda x.\lambda k.M_c)^{\dagger} & = & (\lambda x.\lambda k.\mathcal{N}[\![M_c]\!])
\end{array}$$

$$\begin{array}{lcl}
k^{\ddagger} & = & k \\
(\lambda x.M_c)^{\ddagger} & = & (\lambda x.\mathcal{N}[\![M_c]\!])
\end{array}$$

Assuming that the CPS-code is in the subset determined by $\mathcal{N}$, it closely resembles the target language. The function $\mathcal{B}$ (defined below) maps this subset of the CPS-language to the target language.

$$\begin{array}{lcl}
\mathcal{B}[\![(k\ x)]\!] & = & \texttt{return}\ x \\
\mathcal{B}[\![((\lambda x.M_c)\ c)]\!] & = & \texttt{load}\ c, x; \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![((\lambda z.M_c)\ x)]\!] & = & \texttt{move}\ x, z; \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![((\lambda y.M_c)\ (\lambda x.\lambda k.N_c))]\!] & = & \texttt{makeClosure}\ x, \{\mathcal{B}[\![N_c]\!]\}, y; \\
 & & \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![((c\ x)\ k)]\!] & = & \texttt{instr:}c\ x, z; \texttt{return}\ z \\
\mathcal{B}[\![((y\ x)\ k)]\!] & = & \texttt{tailCall}\ y, x \\
\mathcal{B}[\![(((\lambda w.\lambda k.M_c)\ x)\ k)]\!] & = & \texttt{move}\ x, w; \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![((c\ x)\ (\lambda z.M_c))]\!] & = & \texttt{instr:}c\ x, z; \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![((y\ x)\ (\lambda z.M_c))]\!] & = & \texttt{call}\ y, x, z; \mathcal{B}[\![M_c]\!] \\
\mathcal{B}[\![(((\lambda w.\lambda k.M_c)\ x)\ (\lambda z.N_c))]\!] & = & \texttt{makeClosure}\ w, \{\mathcal{B}[\![M_c]\!]\}, y; \\
 & & \texttt{call}\ y, x, z; \mathcal{B}[\![N_c]\!]
\end{array}$$

The compiler is then the function $\mathcal{T}$ defined below. It is simply the composition of the three stages.

$$\mathcal{T}[\![M]\!] = \mathcal{B}[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!]$$

6. COMPILER CORRECTNESS

Since correct compilers have always been direct, the key to the proof involves a key inductive hypothesis or perhaps a couple of simultaneous hypotheses. I considered the following inductive hypothesis for the compiler presented above.

$$\begin{array}{c}\text{For any } M, Q_c, \rho, \kappa, z \notin \mathsf{fv}(M) \cup \mathsf{fv}(Q_c),\\ \mathcal{D}_s[\![\mathcal{B}[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]Q_c]\!]]\!]]\rho\kappa = \\ \mathcal{D}[\![M]\!]\rho(\lambda v.\mathcal{D}_s[\![\mathcal{B}[\![\mathcal{N}[\![\mathcal{F}[\![z]\!]Q_c]\!]]\!]]\rho[z \mapsto v]\kappa).\end{array}$$

It appears that the induction goes through; however, the proof is long and unwieldy, with subcases, and subcases with subcases, and so on.

Instead, I decided to go with a modular proof. There is a result for each stage of the compiler which establishes that it is correct. The result for each stage is reasonably short and straightforward. When these results are put together, they establish the correctness of the compiler.

The first stage is the CPS-transform. Corollary 1 below establishes the correctness of the CPS-transform; as far as I know, this is the first denotational correctness proof of the CPS-transform. Corollary 1 follows from lemma 1 by letting $Q_c = k$.

LEMMA 1. *For any* M, V, Q_c, ρ, κ,

(i) $(\kappa\ (\mathcal{D}_V[\![V^\star]\!]\rho)) = \mathcal{D}[\![V]\!]\rho\kappa$

(ii) $\mathcal{D}_c[\![\mathcal{F}[\![M]\!]Q_c]\!]\rho\kappa = \mathcal{D}[\![M]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$

COROLLARY 1. *For any* $M, \mathcal{D}_c[\![\mathcal{F}[\![M]\!]k]\!] = \mathcal{D}[\![M]\!]$.

The second stage is normalization of CPS-terms. Corollary 2 below establishes the correctness of normalization of the output of the CPS-transform. It follows from lemma 2 by letting $M_c = \mathcal{F}[\![M]\!]k$.

LEMMA 2. *For any* V_c, Q_c, M_c,

(i) $\mathcal{D}_V[\![V_c]\!] = \mathcal{D}_V[\![V_c^\dagger]\!]$

(ii) $\mathcal{D}_Q[\![Q_c]\!] = \mathcal{D}_Q[\![Q_c^\ddagger]\!]$

(iii) $\mathcal{D}_c[\![M_c]\!] = \mathcal{D}_c[\![\mathcal{N}[\![M_c]\!]]\!]$

COROLLARY 2. *For any* $M, \mathcal{D}_c[\![\mathcal{F}[\![M]\!]k]\!] = \mathcal{D}_c[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!]$.

Corollary 3 establishes that the denotation of the output of both stages one and two is the same as the denotation of the original term. It follows from corollary 1, corollary 2, and the transitive property.

COROLLARY 3. *For any* $M, \mathcal{D}_c[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!] = \mathcal{D}[\![M]\!]$.

The third stage translates a subset of the CPS-language to the target language. Corollary 4 establishes that the denotation of the three stages is the same as the denotation of only the first two stages. It follows from lemma 3 and lemma 4 by letting $M_c = \mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]$.

LEMMA 3. *For any* $M_c, \mathcal{N}[\![M_c]\!] \in \mathsf{domain}(\mathcal{B})$.

LEMMA 4. *For any* $M_c \in \mathsf{domain}(\mathcal{B}), \mathcal{D}_s[\![\mathcal{B}[\![M_c]\!]]\!] = \mathcal{D}_c[\![M_c]\!]$.

COROLLARY 4. *For any* $M, \mathcal{D}_s[\![\mathcal{B}[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!]]\!] = \mathcal{D}_c[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!]$.

Now the correctness theorem is easy. Putting together corollary 3 and corollary 4, we get $\mathcal{D}_s[\![\mathcal{B}[\![\mathcal{N}[\![\mathcal{F}[\![M]\!]k]\!]]\!]]\!] = \mathcal{D}[\![M]\!]$. The result then follows from the definition of $\mathcal{T}$.

THEOREM 1. *For any* $M, \mathcal{D}_s[\![\mathcal{T}[\![M]\!]]\!] = \mathcal{D}[\![M]\!]$.

7. CONCLUSION

I have presented a proof that a simple staged CPS-transform based compiler is correct. The proof has several desirable features:

It is tractable. Each stage has a correctness proof that is bite-sized. Further, the arguments are clear, and proofs rely only on structural induction.

Since one of the stages is the CPS-transform, the proof includes a proof of correctness of the CPS-transform. Typically, the correctness of the CPS-transform is established operationally. The denotational proof is no more complicated, and may have benefits beyond its use in the correctness result presented here.

The proof is modular, and so gains all the advantages of modularity: should there be any errors, they are isolated within a particular component; components can be replaced; and components can be added.

There are some features that are absent in this work. The compiler presented does no optimization. Because of the modular nature of the proof, it should be unproblematic to add a correct optimization stage. All that would be required to extend this proof to a proof of correctness of an optimizing compiler would be a proof of the correctness of the optimizer itself. The source language presented does not have assignment. It would be worthwhile to add assignment to make these proofs a more realistic platform on which to build. The addition of these features is future work.

8. ACKNOWLEDGEMENTS

I would like to thank Melissa Nunes-Harwitt and Richard Zanibbi for carefully reading previous drafts of this paper, and Will Clinger for looking carefully at my proofs.

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley; 2nd edition, 2007.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991.

[3] *CHICKEN — A practical and portable Scheme system*. `http://www.call-with-current-continuation.org/`.

[4] W. D. Clinger. *The Scheme 311 Compiler an Exercise in Denotational Semantics. Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 356–364, 1984.

[5] W. D. Clinger. Common Larceny. *Proceedings of the 2005 International Lisp Conference*, 101–107, 2005.

[6] W. D. Clinger, L. T. Hansen. Lambda, the ultimate label; or a simple optimizing compiler for Scheme. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 128–139, 1994.

[7] O. Danvy, L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science*, Vol. 308, No. 1–3, 239–257, 2003.

[8] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, M. Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, Vol. 12, No. 2, 159–182, 2002.

[9] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen. The Essence of Compiling with Continuations. *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, 237–247, 1993.

[10] D. P. Friedman, M. Wand, C. T. Haynes. *Essentials of Programming Languages*. The MIT Press; 2nd edition, 2001.

[11] A. D. Gordon, P. D. Hankin, S. B. Lassen. Compilation and Equivalence of Imperative Objects. *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, 74–87, 1997.

[12] R. Kelsey, W. Clinger, J. Rees editors. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, 26–76, 1998.

[13] D. A. Kranz, R. A. Kelsey, J. A. Rees, P. Hudak, J. Philbin. ORBIT: An Optimizing Compiler for Scheme. *Proceedings of the SIGPLAN symposium on Compiler contruction*, 219–233, 1986.

[14] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1991.

[15] A. Nunes-Harwitt. Advice about Debugger Construction. *Proceedings of the 2005 International LISP Conference*, 289–298, 2005.

[16] D. P. Oliva. *Advice Concerning the Structuring of Backends*. PhD thesis, 19??, Northeastern University, Boston, MA, USA.

[17] D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP Verified PreScheme Compiler. *Lisp and Symbolic Computation*, Vol. 8, No. 1/2, 111–182, 1995.

[18] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, Vol. 1, 125–159, 1975.

[19] M. Rittri. *Searching Programming Libraries by Type and Proving Compiler Correctness by Bisimulation*. PhD thesis, 1990, University of Göteborg, Göteborg, Sweden.

[20] A. Sabry, P. Wadler. A Reflection on Call-by-Value. *ACM SIGPLAN Notices*, Vol. 31, No. 6, 13–24, 1996.

[21] M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten editors. Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`.

[22] G. L. Steele Jr. *RABBIT: A Compiler for SCHEME*. MIT AI Lab. Technical Report 474, 1978.

[23] M. Wand. Correctness of Procedure Representations in Higher-Order Assembly Language. *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, 294–311, 1991.

[24] P. Winston, B. Horn. *Lisp*. Addison Wesley; 3rd edition, 1989.

APPENDIX
A. PROOFS

Below are the proofs of the key theorems.

Proof of lemma 1

By simultaneous induction on parts (i) and (ii).

Part (i):

- Suppose $V = c$.
  $(\kappa\ \mathcal{D}_V[\![c^\star]\!]\rho)$
  $= (\kappa\ \mathcal{K}(c))$
  $= (\lambda\rho'.\lambda\kappa'.(\kappa'\ \mathcal{K}(c)))\rho\kappa$
  $= \mathcal{D}[\![c]\!]\rho\kappa$
- Suppose $V = x$.
  $(\kappa\ \mathcal{D}_V[\![x^\star]\!]\rho)$
  $= (\kappa\ \rho(x))$
  $= (\lambda\rho'.\lambda\kappa'.(\kappa'\ \rho'(x)))\rho\kappa$
  $= \mathcal{D}[\![x]\!]\rho\kappa$
- Suppose $V = (\lambda x.M)$.
  $(\kappa\ \mathcal{D}_V[\![(\lambda x.M)^\star]\!]\rho)$
  $= (\kappa\ \mathcal{D}_V[\![(\lambda x.\lambda k.\mathcal{F}[\![M]\!]k)]\!]\rho)$
  $= (\kappa\ (\lambda v.\mathcal{D}_c[\![\mathcal{F}[\![M]\!]k]\!]\rho[x \mapsto v]))$
  $= (\kappa\ (\lambda v.\lambda\kappa'.\mathcal{D}_c[\![\mathcal{F}[\![M]\!]k]\!]\rho[x \mapsto v]\kappa'))$
  $= (\kappa\ (\lambda v.\lambda\kappa'.\mathcal{D}[\![M]\!]\rho[x \mapsto v](\mathcal{D}_Q[\![k]\!]\rho[x \mapsto v]\kappa')))$
  $= (\kappa\ (\lambda v.\lambda\kappa'.\mathcal{D}[\![M]\!]\rho[x \mapsto v]\kappa'))$
  $= (\kappa\ (\lambda v.\mathcal{D}[\![M]\!]\rho[x \mapsto v]))$
  $= (\lambda\rho'.\lambda\kappa'.(\kappa'\ (\lambda v.\mathcal{D}[\![M]\!]\rho'[x \mapsto v])))\rho\kappa$
  $= \mathcal{D}[\![(\lambda x.M)]\!]\rho\kappa$

Part (ii):

- Suppose $M = V$.
  $\mathcal{D}_c[\![\mathcal{F}[\![V]\!]Q_c]\!]\rho\kappa$
  $= \mathcal{D}_c[\![(Q_c\ V^\star)]\!]\rho\kappa$
  $= ((\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)\ (\mathcal{D}_V[\![V^\star]\!]\rho))$
  $= \mathcal{D}[\![V]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$

- Suppose $M = (V\ V')$.
  $\mathcal{D}_c[\![\mathcal{F}[\![(V\ V')]\!]Q_c]\!]\rho\kappa$
  $= \mathcal{D}_c[\![((V^\star\ V'^\star)\ Q_c)]\!]\rho\kappa$
  $= (((\mathcal{D}_V[\![V^\star]\!]\rho)\ (\mathcal{D}_V[\![V'^\star]\!]\rho))\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))$
  $= ((\lambda n.(((\mathcal{D}_V[\![V^\star]\!]\rho)\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)))\ (\mathcal{D}_V[\![V'^\star]\!]\rho))$
  $= \mathcal{D}[\![V']\!]\rho(\lambda n.(((\mathcal{D}_V[\![V^\star]\!]\rho)\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)))$
  $= ((\lambda m.\mathcal{D}[\![V']\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))\ (\mathcal{D}_V[\![V^\star]\!]\rho))$
  $= \mathcal{D}[\![V]\!]\rho(\lambda m.\mathcal{D}[\![V']\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))$
  $= \mathcal{D}[\![(V\ V')]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$
- Suppose $M = (V\ P)$.
  $\mathcal{D}_c[\![\mathcal{F}[\![(V\ P)]\!]Q_c]\!]\rho\kappa$
  $= \mathcal{D}_c[\![\mathcal{F}[\![P]\!](\lambda x.((V^\star\ x)\ Q_c))]\!]\rho\kappa$
  $= \mathcal{D}[\![P]\!]\rho(\mathcal{D}_Q[\![(\lambda x.((V^\star\ x)\ Q_c))]\!]\rho\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda n.\mathcal{D}_c[\![((V^\star\ x)\ Q_c)]\!]\rho[x \mapsto n]\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda n.(((\mathcal{D}_V[\![V^\star]\!]\rho[x \mapsto n])\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho[x \mapsto n]\kappa)))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda n.(((\mathcal{D}_V[\![V^\star]\!]\rho)\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)))$
  $= ((\lambda m.\mathcal{D}[\![P]\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))\ (\mathcal{D}_V[\![V^\star]\!]\rho))$
  $= \mathcal{D}[\![V]\!]\rho(\lambda m.\mathcal{D}[\![P]\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))$
  $= \mathcal{D}[\![(V\ P)]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$
- Suppose $M = (P\ V)$.
  $\mathcal{D}_c[\![\mathcal{F}[\![(P\ V)]\!]Q_c]\!]\rho\kappa$
  $= \mathcal{D}_c[\![\mathcal{F}[\![P]\!](\lambda y.((y\ V^\star)\ Q_c))]\!]\rho\kappa$
  $= \mathcal{D}[\![P]\!]\rho(\mathcal{D}_Q[\![(\lambda y.((y\ V^\star)\ Q_c))]\!]\rho\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}_c[\![((y\ V^\star)\ Q_c)]\!]\rho[y \mapsto m]\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.((m\ (\mathcal{D}_V[\![V^\star]\!]\rho[y \mapsto m]))\ (\mathcal{D}_Q[\![Q_c]\!]\rho[y \mapsto m]\kappa)))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.((m\ (\mathcal{D}_V[\![V^\star]\!]\rho))\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.((\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)))\ (\mathcal{D}_V[\![V^\star]\!]\rho)))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}[\![V]\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))$
  $= \mathcal{D}[\![(P\ V)]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$
- Suppose $M = (P\ P')$.
  $\mathcal{D}_c[\![\mathcal{F}[\![(P\ P')]\!]Q_c]\!]\rho\kappa$
  $= \mathcal{D}_c[\![\mathcal{F}[\![P]\!](\lambda y.\mathcal{F}[\![P']\!](\lambda x.((y\ x)\ Q_c)))]\!]\rho\kappa$
  $= \mathcal{D}[\![P]\!]\rho(\mathcal{D}_Q[\![(\lambda y.\mathcal{F}[\![P']\!](\lambda x.((y\ x)\ Q_c)))]\!]\rho\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}_c[\![\mathcal{F}[\![P']\!](\lambda x.((y\ x)\ Q_c))]\!]\rho[y \mapsto m]\kappa)$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}[\![P']\!]\rho[y \mapsto m](\mathcal{D}_Q[\![(\lambda x.((y\ x)\ Q_c))]\!]\rho[y \mapsto m]\kappa))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}[\![P']\!]\rho(\lambda n.\mathcal{D}_c[\![((y\ x)\ Q_c)]\!]\rho[y \mapsto m][x \mapsto n]\kappa))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}[\![P']\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho[y \mapsto m][x \mapsto n]\kappa))))$
  $= \mathcal{D}[\![P]\!]\rho(\lambda m.\mathcal{D}[\![P']\!]\rho(\lambda n.((m\ n)\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))))$
  $= \mathcal{D}[\![(P\ P')]\!]\rho(\mathcal{D}_Q[\![Q_c]\!]\rho\kappa)$

QED
Proof of lemma 2

By simultaneous induction on parts (i), (ii), and (iii).
Part (i):

- Suppose $V_c = W$.
  Since $W^\dagger = W$, it follows immediately that $\mathcal{D}_V[\![W^\dagger]\!] = \mathcal{D}_V[\![W]\!]$.
- Suppose $V_c = (\lambda x.\lambda k.M_c)$.
  $\mathcal{D}_V[\![(\lambda x.\lambda k.M_c)]\!]$
  $= \lambda\rho.(\lambda v.\mathcal{D}_c[\![M_c]\!]\rho[x \mapsto v])$
  $= \lambda\rho.(\lambda v.\mathcal{D}_c[\![\mathcal{N}[\![M_c]\!]]\!]\rho[x \mapsto v])$
  $= \mathcal{D}_V[\![(\lambda x.\lambda k.\mathcal{N}[\![M_c]\!])]\!]$
  $= \mathcal{D}_V[\![(\lambda x.\lambda k.M_c)^\dagger]\!]$

Part (ii):

- Suppose $Q_c = k$.
  Since $k^\ddagger = k$, , it follows immediately that $\mathcal{D}_Q[\![k^\ddagger]\!] = \mathcal{D}_Q[\![k]\!]$.
- Suppose $Q_c = (\lambda x.M_c)$.
  $\mathcal{D}_Q[\![(\lambda x.M_c)]\!]$
  $= \lambda\rho.\lambda\kappa.(\lambda v.\mathcal{D}_c[\![M_c]\!]\rho[x \mapsto v]\kappa)$
  $= \lambda\rho.\lambda\kappa.(\lambda v.\mathcal{D}_c[\![\mathcal{N}[\![M_c]\!]]\!]\rho[x \mapsto v]\kappa)$
  $= \mathcal{D}_Q[\![(\lambda x.\mathcal{N}[\![M_c]\!])]\!]$
  $= \mathcal{D}_Q[\![(\lambda x.M_c)^\ddagger]\!]$

Part (iii):

- Suppose $M_c = (k\ x)$.
  Since $\mathcal{N}[\![(k\ x)]\!] = (k\ x)$, it follows immediately that $\mathcal{D}_c[\![\mathcal{N}[\![(k\ x)]\!]]\!] = \mathcal{D}_c[\![(k\ x)]\!]$.
- Suppose $M_c = (k\ R_c)$.
  $\mathcal{D}_c[\![(k\ R_c)]\!]$
  $= \lambda\rho.\lambda\kappa.((\lambda v.(\kappa\ v))\ (\mathcal{D}_V[\![R_c]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![(k\ x)]\!]\rho[x \mapsto v]\kappa)\ (\mathcal{D}_V[\![R_c]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![(k\ x)]\!]\rho[x \mapsto v]\kappa)\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))$
  $= \mathcal{D}_c[\![((\lambda x.(k\ x))\ R_c^\dagger)]\!]$
  $= \mathcal{D}_c[\![\mathcal{N}[\![(k\ R_c)]\!]]\!]$
- Suppose $M_c = ((\lambda x.N_c)\ V_c)$.
  $\mathcal{D}_c[\![((\lambda x.N_c)\ V_c)]\!]$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[x \mapsto v]\kappa)\ (\mathcal{D}_V[\![V_c]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![\mathcal{N}[\![N_c]\!]]\!]\rho[x \mapsto v]\kappa)\ (\mathcal{D}_V[\![V_c^\dagger]\!]\rho))$
  $= \mathcal{D}_c[\![((\lambda x.\mathcal{N}[\![N_c]\!])\ V_c^\dagger)]\!]$
  $= \mathcal{D}_c[\![\mathcal{N}[\![((\lambda x.N_c)\ V_c)]\!]]\!]$
- Suppose $M_c = ((V_c\ x)\ Q_c)$.
  $\mathcal{D}_c[\![((V_c\ x)\ Q_c)]\!]$
  $= \lambda\rho.\lambda\kappa.(((\mathcal{D}_V[\![V_c]\!]\rho)\ (\mathcal{D}_V[\![x]\!]\rho))\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))$
  $= \lambda\rho.\lambda\kappa.(((\mathcal{D}_V[\![V_c^\dagger]\!]\rho)\ (\mathcal{D}_V[\![x]\!]\rho))\ (\mathcal{D}_Q[\![Q_c^\ddagger]\!]\rho\kappa))$
  $= \mathcal{D}_c[\![((V_c^\dagger\ x)\ Q_c^\ddagger)]\!]$
  $= \mathcal{D}_c[\![\mathcal{N}[\![((V_c\ x)\ Q_c)]\!]]\!]$
- Suppose $M_c = ((V_c\ R_c)\ Q_c)$.
  $\mathcal{D}_c[\![((V_c\ R_c)\ Q_c)]\!]$
  $= \lambda\rho.\lambda\kappa.(((\mathcal{D}_V[\![V_c]\!]\rho)\ (\mathcal{D}_V[\![R_c]\!]\rho))\ (\mathcal{D}_Q[\![Q_c]\!]\rho\kappa))$
  $= \lambda\rho.\lambda\kappa.(((\mathcal{D}_V[\![V_c^\dagger]\!]\rho)\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))\ (\mathcal{D}_Q[\![Q_c^\ddagger]\!]\rho\kappa))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.(((\mathcal{D}_V[\![V_c^\dagger]\!]\rho)\ v)\ (\mathcal{D}_Q[\![Q_c^\ddagger]\!]\rho\kappa)))\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.(((\mathcal{D}_V[\![V_c^\dagger]\!]\rho)\ \mathcal{D}_V[\![x]\!]\rho[x \mapsto v])$
  $\qquad (\mathrm{D}_Q[\![Q_c^\ddagger]\!]\rho\kappa)))\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.(((\mathcal{D}_V[\![V_c^\dagger]\!]\rho[x \mapsto v])\ \mathcal{D}_V[\![x]\!]\rho[x \mapsto v])$
  $\qquad (\mathrm{D}_Q[\![Q_c^\ddagger]\!]\rho[x \mapsto v]\kappa)))\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![((V_c^\dagger\ x)\ Q_c^\ddagger)]\!]\rho[x \mapsto v]\kappa)\ (\mathcal{D}_V[\![R_c^\dagger]\!]\rho))$
  $= \mathcal{D}_c[\![((\lambda x.((V_c^\dagger\ x)\ Q_c^\ddagger))\ R_c^\dagger)]\!]$
  $= \mathcal{D}_c[\![\mathcal{N}[\![((V_c\ R_c)\ Q_c)]\!]]\!]$

QED
Proof of lemma 4

By structural induction on the domain of $\mathcal{B}$.

- Suppose $M_c = (k\ x)$.
  $\mathcal{D}_s[\![\mathcal{B}[\![(k\ x)]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{return}\ x]\!]$
  $= \lambda\rho.\lambda\kappa.(\kappa\ \rho(x))$
  $= \mathcal{D}_c[\![(k\ x)]\!]$
- Suppose $M_c = ((\lambda x.N_c)\ c)$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((\lambda x.N_c)\ c)]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{load}\ c, x; \mathcal{B}[\![N_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[x \mapsto \mathcal{K}(c)]\kappa$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_c[\![N_c]\!]\rho[x \mapsto \mathcal{K}(c)]\kappa$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[x \mapsto v]\kappa)\ \mathcal{K}(c))$
  $= \mathcal{D}_c[\![((\lambda x.N_c)\ c)]\!]$
- Suppose $M_c = ((\lambda z.N_c)\ x)$.

$\mathcal{D}_s[\![\mathcal{B}[\![((\lambda z.N_c)\ x)]\!]]\!]$
$= \mathcal{D}_s[\![\texttt{move}\ x, z; \mathcal{B}[\![N_c]\!]]\!]$
$= \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[z \mapsto \rho(x)]\kappa$
$= \lambda\rho.\lambda\kappa.\mathcal{D}_c[\![N_c]\!]\rho[z \mapsto \rho(x)]\kappa$
$= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[z \mapsto v]\kappa)\ \rho(x))$
$= \mathcal{D}_c[\![((\lambda z.N_c)\ x)]\!]$

- Suppose $M_c = ((\lambda y.N_c)\ (\lambda x.\lambda k.N'_c))$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((\lambda y.N_c)\ (\lambda x.\lambda k.N'_c))]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{makeClosure}\ x, \{N'_c\}, y; \mathcal{B}[\![N_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[y \mapsto (\lambda u.\mathcal{D}_s[\![\mathcal{B}[\![N'_c]\!]]\!]\rho[x \mapsto u])]\kappa$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_c[\![N_c]\!]\rho[y \mapsto (\lambda u.\mathcal{D}_c[\![N'_c]\!]\rho[x \mapsto u])]\kappa$
  $= \lambda\rho.\lambda\kappa.((\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[y \mapsto v]\kappa)\ (\lambda u.\mathcal{D}_c[\![N'_c]\!]\rho[x \mapsto u]))$
  $= \mathcal{D}_c[\![((\lambda y.N_c)\ (\lambda x.\lambda k.N'_c))]\!]$
- Suppose $M_c = ((c\ x)\ k)$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((c\ x)\ k)]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{instr:}c\ x, z; \texttt{return}\ z]\!]$
  $= \lambda\rho.\lambda\kappa.((\mathcal{K}(c)\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![\texttt{return}\ z]\!]\rho[z \mapsto v]\kappa))$
  $= \lambda\rho.\lambda\kappa.((\mathcal{K}(c)\ \rho(x))\ (\lambda v.(\kappa\ v)))$
  $= \mathcal{D}_c[\![((c\ x)\ k)]\!]$
- Suppose $M_c = ((y\ x)\ k)$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((y\ x)\ k)]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{tailCall}\ y, x]\!]$
  $= \lambda\rho.\lambda\kappa.((\rho(y)\ \rho(x))\ \kappa)$
  $= \mathcal{D}_c[\![((y\ x)\ k)]\!]$
- Suppose $M_c = (((\lambda w.\lambda k.N_c)\ x)\ k)$.
  $\mathcal{D}_s[\![\mathcal{B}[\![(((\lambda w.\lambda k.N_c)\ x)\ k)]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{move}\ x, w; \mathcal{B}[\![N_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[w \mapsto \rho(x)]\kappa$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_c[\![N_c]\!]\rho[w \mapsto \rho(x)]\kappa$
  $= \lambda\rho.\lambda\kappa.(((\lambda u.\mathcal{D}_c[\![N_c]\!]\rho[w \mapsto u])\ \rho(x))\ \kappa)$
  $= \mathcal{D}_c[\![(((\lambda w.\lambda k.N_c)\ x)\ k)]\!]$
- Suppose $M_c = ((c\ x)\ (\lambda z.N_c))$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((c\ x)\ (\lambda z.N_c))]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{instr:}c\ x, z; \mathcal{B}[\![N_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.((\mathcal{K}(c)\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[z \mapsto v]\kappa))$
  $= \lambda\rho.\lambda\kappa.((\mathcal{K}(c)\ \rho(x))\ (\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[z \mapsto v]\kappa))$
  $= \mathcal{D}_c[\![((c\ x)\ (\lambda z.N_c))]\!]$
- Suppose $M_c = ((y\ x)\ (\lambda z.N_c))$.
  $\mathcal{D}_s[\![\mathcal{B}[\![((y\ x)\ (\lambda z.N_c))]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{call}\ y, x, z; \mathcal{B}[\![N_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.((\rho(y)\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[z \mapsto v]\kappa))$
  $= \lambda\rho.\lambda\kappa.((\rho(y)\ \rho(x))\ (\lambda v.\mathcal{D}_c[\![N_c]\!]\rho[z \mapsto v]\kappa))$
  $= \mathcal{D}_c[\![((y\ x)\ (\lambda z.N_c))]\!]$
- Suppose $M_c = (((\lambda w.\lambda k.N_c)\ x)\ (\lambda z.N'_c))$.
  $\mathcal{D}_s[\![\mathcal{B}[\![(((\lambda w.\lambda k.N_c)\ x)\ (\lambda z.N'_c))]\!]]\!]$
  $= \mathcal{D}_s[\![\texttt{makeClosure}\ w, \{\mathcal{B}[\![N_c]\!]\}, y; \texttt{call}\ y, x, z; \mathcal{B}[\![N'_c]\!]]\!]$
  $= \lambda\rho.\lambda\kappa.\mathcal{D}_s[\![\texttt{call}\ y, x, z; \mathcal{B}[\![N'_c]\!]]\!]\rho[y \mapsto (\lambda u.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[w \mapsto u])]\kappa$
  $= \lambda\rho.\lambda\kappa.(((\lambda u.\mathcal{D}_s[\![\mathcal{B}[\![N_c]\!]]\!]\rho[w \mapsto u])\ \rho(x))\ (\lambda v.\mathcal{D}_s[\![\mathcal{B}[\![N'_c]\!]]\!]\rho[z \mapsto v]\kappa))$
  $= \lambda\rho.\lambda\kappa.(((\lambda u.\mathcal{D}_c[\![N_c]\!]\rho[w \mapsto u])\ \rho(x))\ (\lambda v.\mathcal{D}_c[\![N'_c]\!]\rho[z \mapsto v]\kappa))$
  $= \mathcal{D}_s[\![(((\lambda w.\lambda k.N_c)\ x)\ (\lambda z.N'_c))]\!]$

QED